Model-driven Engineering for Requirements Analysis

Benoit Baudry
IRISA & INRIA
Campus universitaire de Beaulieu
35042 Rennes cedex, France
bbaudry@irisa.fr

Clémentine Nebut
LIRMM, CNRS & univ. Montpellier 2
161, rue Ada
34392 Montpellier cedex 5, France
nebut@lirmm.fr

Yves Le Traon
ENSTB & IRISA
2, rue de la Châtaigneraie
35576 Cesson Sévigné, France
Yves.LETRAON@enst-bretagne.fr

Abstract

Requirements engineering (RE) encompasses a set of activities for eliciting, modelling, agreeing, communicating and validating requirements that precisely define the problem domain for a software system. Several tools and methods exist to perform each of these activities, but they mainly remain separate, making it difficult to capture the global consistency of large requirement documents. In this paper we introduce model-driven engineering (MDE) as a possible technical solution to integrate these activities in a common framework. First, we dicuss how RE can leverage the two main techniques for MDE: metamodelling and model transformation. Then, we introduce a metamodel for requirements and present how we have implemented this metamodel to make it executable and usable through a constrained natural language for requirements definition.

1 Introduction

A crucial issue when starting a new software development project consists in eliciting, defining, modelling and agreeing on the requirements for the system. This requires a lot of effort, involving all stakeholders related to the project and managing a large amount of information. All these activities have been intensively investigated by the requirements engineering (RE) academic and industrial community. Today, a number of tools, environments and solid theoretical knowledge have been produced for RE. However, it remains a composite activity and sub-activities are still very much disconnected from each others. This makes it very difficult to check the consistency between numerous documents, impact local changes on a large set of requirements or have a global understanding of the requirements.

Model-driven engineering (MDE) offers a technical framework that can relate software developement activities around metamodels and model transformations, and we believe that it can be similarly used to relate RE activities. MDE advocates the use of models as first-class entities for software development. This means first that models have to be more than drawings and must be formally defined and automatically computable by programs. This first step is achieved through the definition of metamodels that formally and completely define models. The metamodel describes the structure of models and can be extended with operations that specify the operational semantics of models. Second, this means that it is necessary to define, specify and implement programs that process these models. This type of program is called a model transformation. This is a powerful mechanism to automate a number of development activities: refinement, refactoring, translation in another modelling language, code generation, etc.

The core contribution of this paper is the definition of a metamodelling environment for requirements modelling and simulation. This work has been initiated in two collaborations with industrial partners (THALES [11] and France Telecom). In order to produce efficient test cases from the requirements, we had to disambiguate the functional requirements and perform requirements analysis. To design a flexible test environment, we use MDE and define a metamodel for the concepts we need at requirement level. This metamodel captures functional requirements as use cases with pre and post conditions that constrain the activation of the use cases. Thanks to executable metamodelling, we can add operations in this metamodel in order to simulate the

requirements model. Simulation is very useful to validate the completeness, the consistency of requirements as well as the business logic.

As we will discuss it in this paper, a major benefit of this MDE-based approach for requirement analysis is to allow interoperability between the several RE tasks like modelling, understandability and elicitation. Another benefit of MDE for requirements engineering is to improve the integration of RE with subsequent model-driven software developement steps. As Sommerville points out in [18], RE and software engineering are still two very disctinct processes. The integration of these activities is a major issue to deal with continuous requirements changes and to integrate RE in a spiral development cycle.

The paper is organized as follows. Section 2 recalls the MDE approach and introduces the techniques used for experiments. The rest of the paper presents how we have applied these techniques for requirements analysis and simulation. Section 3 presents the requirements metamodel that captures the concepts of use cases with contracts and a data model. Section 4 introduces the execution semantics of our requirements metamodel and details how we have extended the metamodel to implement this semantics in order to have simulable models. At last, we present related works and conclude.

2 Model-Driven Engineering

Model-Driven Engineering (MDE) is an approach to software development that focuses on models as first class entities for development (as opposed to programs). Models can describe various concerns such as functionality, time constraints, security, maintainability etc. MDE emphasizes the need to have productive models that can be automatically manipulated by programs. To make the models productive, it is necessary to completely and formally define them. In the MDE context, metamodels are used to build this formal definition. Based on the definition of a metamodel, it is possible to implement model transformations that automatically refine, compose, refactor or reverse models.

Metamodels and automatic model transformations are two crucial mechanisms for MDE. In this section we detail metamodelling and discuss how operations can be added in metamodels to enable the simulation of models. We also introduce the Kermeta metamodelling language that is used in our works to implement metamodels and transformations.

2.1 Executable Metamodelling

Metamodelling consists in building a metamodel for domain specific languages. These metamodels are defined with metamodelling languages like MOF [15], EMOF [15]

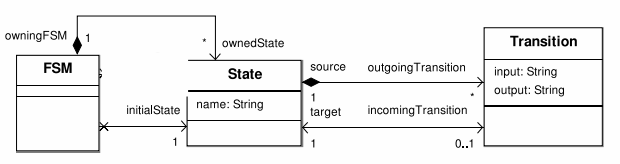

Figure 1. Structure for the FSM metamodel

or Ecore [5]. These languages enable the definition of domain specific concepts and the relationships between theses concepts (association, composition, specialization). There exist tools that can then check that models, instances of a metamodel, conform to the structure defined by the metamodel. This consists in checking that the model is a set of objects which super class is defined in the metamodel, and the relationships between the objects conform to the relationships defined in the metamodel. Currently, most metamodels are defined with these languages and thus define only the structure of the models (what concepts they can use and how they can be related). For example, Figure 1 describes the structure for finite state machines: a FSM is composed of a set of states that have a set of outgoing and incoming transitions. However, such a definition lacks information about the semantics of FSM.

It is possible to use constraint languages such like the OCL [16] to add semantic constraints on the structure of the metamodels. For example, in Figure 1, it would be possible to use OCL to constrain the initial state to be included in the set `ownedStates` of the FSM. However, the constraints define static semantics and OCL is not meant to be used for defining the operational semantics of the models.

In order to improve the definition of metamodels, it is necessary to add actions. This is called executable metamodelling. The models that are instances of the metamodel can be executed which enables the validation of the operational semantics through simulation. For example, Figure 2 defines operations in the FSM metamodel. With an action language, it is possible to define the body of these operations. For example, write the sequence of actions that specifies the `step()` operation: if an input event matches the input event of an outgoing transition for the current state, then the transition can be triggered and the current state is updated. It is then possible to create a FSM that conforms to this metamodel and call the step run operation on the instance of the FSM class to simulate the FSM.

The main benefit of executable modelling is that a model that conforms to the metamodel is executable by construction. In other words, it is not necessary to translate/compile the model into another executable language to simulate its behaviour. This is a valuable mechanism both for the modellers and the metamodellers. For the modellers, this

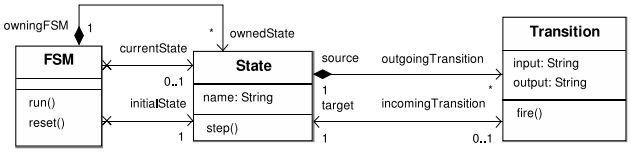

Figure 2. Executable FSM metamodel

provides a rapid feedback to validate the models. For the metamodellers, this simplifies the fine-tuning of the operational semantics of a modelling language: when simulating a model, if the behaviour does not conform to the expectations of the metamodellers, it is possible to modify the operation in the metamodel and directly run the same model again. This gives immediate feedback to the metamodellers about the changes they made.

2.2 Kermeta Metamodelling Language

Kermeta [9] is an open source metamodelling environment that has been designed as an extension to the metadata language EMOF [15] with an action language for specifying semantics and behaviour of meta-models. The action language is imperative and object-oriented and is used to provide an implementation of operations defined in metamodels. A more detailed description of the language is presented in [12].

The Kermeta action language has been specially designed to process models. It includes both OO features and model specific features. Convenient constructions of the Object Constraint Language (OCL) such as closures (e.g. each, collect, select) are also available in Kermeta. The action language offered by Kermeta is well adapted to model-oriented activities such as:

- specification of abstract syntax, static semantics (with the support for OCL) and dynamic semantics,
- model and metamodel simulation and prototyping,
- model transformation,
- aspect weaving.

For example, Figure 3 displays the definition of the `step` operation with the Kermeta language in the FSM metamodel.

In this work, we use Kermeta both to add operations in our requirements metamodel and to define a sequence of model transformations from an input textual language towards this metamodel.

```
operation step(c : String) : String

 var validTransitions : Collection<
     Transition>
 validTransitions :=     outgoingTransition.
     select { t |
     t.input.equals(c)
 }
 if validTransitions.size > 1 then
  raise NonDeterminism.new
 end
 result := validTransitions.one.fire
```

Figure 3. Kermeta definition of the `step` operation

3 A metamodelling environment for requirements analysis

To experiment the application of MDE techniques for requirements engineering, we have developed a prototype around an executable requirements metamodel. This metamodel is the core element for our experiments. It is defined from the experience we had with modelling requirements for THALES and France Telecom. The metamodel specifically targets the definition, analysis and validation of functional requirements that define sequences of service activations. These requirements are expressed as use cases associated with pre and post conditions. The pre-condition defines the conditions in which a use case can be executed and the post condition expresses the effect the use case has on the state of the system. The requirements also manipulate data (the business concepts) that have to be part of the model. The metamodel presented here captures the different concepts needed to model these requirements.

We illustrate the metamodel using examples from a Library Management System(LMS) which requirements are described below:

- A library is maintained by a librarian.
- The librarian can register new books in the LMS and can also register books that have been fixed.
- The librarian can register customers.
- A customer must register in the library to avail the facility of borrowing the books.
- Books must be registered before they are available to the customers.

- The customer can borrow books if they are available and not damaged.
- When a customer returns the book, the book is not available for any customer to borrow again, till the librarian performs an inventory check.

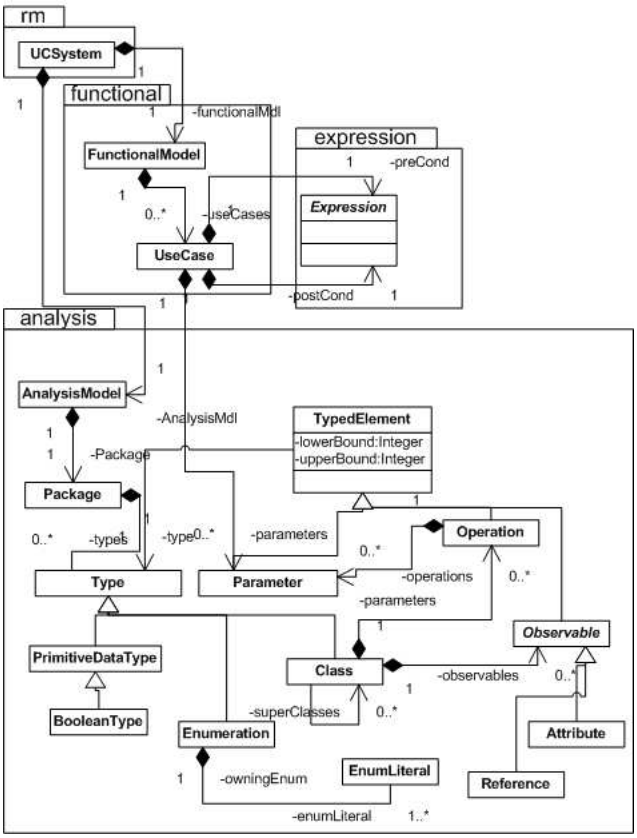

Figure 4. Requirement metamodel overview

Figure 4 displays a general view of the metamodel. `UCSystem` is the root class for the metamodel and is composed of the two main parts of the model i.e. the FUNCTIONALMODEL and the `AnalysisModel`. The first one specifies use cases with contracts and the latter the static analysis data model for the requirements. This metamodel captures the dynamic part of the requirements, but is not sufficient to capture the static part and in particular the handled data. We are aware that the metamodel should be extended for a better support of requirements, however the work presented here yet provides significant benefits, in particular using the simulation process explained in the next section.

The `UseCase` class represents a use case in our model and each use case has exactly one pre-condition and one post-condition. Contracts allow the system designer to specify both when a use case is applicable (precondition), and the effect of a use case execution on the system's state (post-condition). Contracts, represented by the `Expression` class in our metamodel, are expressed as first order logical expressions having a set of typed parameters, combined with different logical operators. All the necessary concepts are present in the metamodel as sub-classes of `Expression`, but are not represented here to limit the size of the figure. These expressions are used to describe the properties of the system (an actor state, a business concept state etc) at any state during the simulation. These expressions are Boolean expression, thus can be either true or false. Logical operator includes conjunction (and), disjunction (or), negation (not) and implication. In order to increase the expressiveness, exists and forall quantifiers are also included.

At requirements level, a use case mainly depends on the actors involved and business concepts which it has to handle. In our metamodel we treat actors and business concepts as data that can be passed as parameters to the use case. The `Parameter` class in the `analysis` package represents this concept. For example, let us consider the use case borrow of the Library Management System.

```
Use Case borrow(c: customer, b: book)
```

The parameters of this use case are the customer who wants to borrow the book, and the book to be borrowed. Here `customer` is an actor and `book` is a business concept.

The `analysis` package models a high-level analysis data model of the system. It defines the concept of class with operations and attributes. This package is close to the UML class diagram metamodel. A `Class` represents an actor or a business concept. It is composed of a set of `Attributes` and `Operations`. Boolean represented by `BooleanType` class and enumeration represented by `Enumeration` class are supported as primitive data types. There is not support for handling complex data types.

The `Borrow` use case for a Library Management System requires that a customer who wants to borrow the book must be registered and the book she/he wants to borrow must be available and not marked as damaged. After performing the use case `Borrow`, the book is not available and the customer has borrowed the book.

```
Use Case Borrow (b: Book, c: Customer)
Pre: registered(c) and available (b) and
    not damaged (b)
Post: registered(c) and not available (b)
    and borrowed (c, b)
```

The model for the above use case can be captured in a model that conforms to the metamodel defined here (and given in Figure 4). However, it might be difficult to express

requirements directly in the form of use cases. It is especially difficult to define all the contracts at once. Moreover, it is difficult to estimate if the use cases globally express the expected requirements. In order to validate the global set of requirements, we have added a simulation capability in the metamodel (section 4). To facilitate the definition of requirements, we have defined a constrained natural language that enables the definition of requirements as sentences, but that is not presented here.

4 Simulation of the requirements model

Interactive simulation of use cases is a useful way to determine the behavior and correctness of requirements at an early stage of software development process. The requirement analyst can verify whether the requirements model conforms to the system specification. Using a simulation tool allows inconsistencies in contracts and underspecification errors in the requirements to be detected. Properties of the system like invariants can also be verified using model checking techniques.
The simulation technique has been proposed in [13] for test generation purpose and was implemented in Java. For the sake of remembrance we summarize the principles for simulation in section 4.1. Then, section 4.2 details how we now leverage executable metamodelling to simulate the requirements model. We explain how we extend the metamodel of figure 4 to add execution semantics.

4.1 Principles of the simulation

The intuition behind the simulation is to instantiate the use cases, replacing the formal parameters with actual values defined in an initial configuration. We thus need to know all the business entities present in the system for one particular simulation. For example, to deal with two books and two customers, it is necessary to declare (in a RDL file):

```
b1, b2: book
c1, c2: customer
```

The possible instantiations of the use case `borrow (b: book, c: customer)` are then `borrow(b1,c1)`, `borrow(b1,c2)`, `borrow(b2,c1)` and `borrow(b2,c2)`. The instantiated use cases are interactively executed, if their preconditions are satisfied. More formally, the simulation consists in on-the-fly building of a transition system named Use Case Transition System (UCTS). A UCTS is defined by a quadruple (Q, q_0, A, $\hookrightarrow$) where:

- Q is a finite non-empty set of states, each state being defined as a set of instantiated expressions,
- q_0 is the initial state,

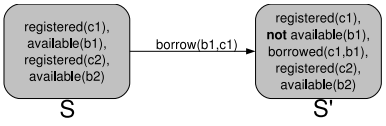

Figure 5. Transition example in the UCTS

- A is the alphabet of actions, an action being an instantiated use case,
- $\hookrightarrow \subseteq Q \times A \times Q$ is the transition function.

A state of the UCTS represents the system's state at a given stage of execution, in terms of values of the defined logical expressions. Each transition is labeled with an instantiated use case, and represents the execution of this instantiated use case. A transition labeled with an instantiated use case `iuc` exists between two states `A` and `B` iff the precondition of `iuc` is satisfied by State `A`, i.e. if `A` logically implies the precondition of `iuc`. The execution of `iuc` leads to State B, which corresponds to the state `A` modified according to the post-condition of `iuc`.
To illustrate this simulation, let us focus on the UCTS excerpt given in Figure 5. From the current state `S`, when we apply the instantiated use case `borrow (b1, c1)`, the new current state is `S'`. To be able to compute the new current state, we have restricted the usage of the postconditions: the postconditions must be deterministic. This restriction is a limitation, however conditional postconditions can still be expressed, making the condition explicit. As an example, let us consider a use case U(x:X) resulting in the predicate p1 or the predicate p2, depending on a given condition c(x). We do not accept the postcondition "p1 or p2" since it is not deterministic: the condition c(x) does not appear in the post- condition. However, the postcondition can be expressed as follows: "c(x)@pre implies p1 and not c(x)@pre implies p2"[1] We have made explicit in this latter postcondition the condition c(x), the postcondition is thus valid. Simulating the system interactively builds part of the corresponding UCTS. For that purpose, we also need an initial state defining the values of the logical expressions defined in the requirements at the initial stage of the simulation.

4.2 The executable requirements metamodel

In this section we detail how we use executable metamodelling to implement simulation directly in the requirements metamodel. This allows us to simulate the require-

[1]The suffix @pre positionned after a predicate in a post-condition means : the value of the predicate before the execution of the use case, this principle with this syntax is taken from the OCL and Eiffel.

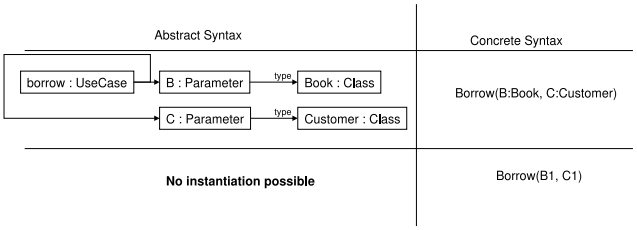

Figure 6. Instantiation issue for use cases

ments and thus to validate the consistency, the completeness and the business logic described in the textual documents. These extensions consist mainly in adding the `simulation` and `instances` packages. First, we detail the need for the `instances` package, then we explain the operations defined in `simulation`. In addition to these packages, we also add several operations in existing classes.

When building a model that conforms to the requirements metamodel (fig. 4), it is possible to define use cases (as instances of the `UseCase` metaclass). For example, figure 6 displays an excerpt of an instance of the requirements metamodel that corresponds to the definition of a use case with two parameters. The right part of the figure gives the concrete syntax for this example. As stated earlier, the simulation manipulates use cases instances. Figure 6 displays the concrete syntax representation of an instance of the borrow use case that we would like to manipulated for simulation. However, EMOF defines only three levels for metamodelling: EMOF, a metamodel and an instance of the metamodel. Thus, it is not possible, in this technological context, to instantiate a use case. As it is shown in Figure 6, the use case definition cannot be instantiated because this definition is an instance of the requirements metamodel. As such, a use case definition is the lowest meta-level that is allowed by MOF.

Since it is not possible to build use case instances, it is necessary to extend the requirements metamodels with this concept in order to implement simulation. That is introduced in the `instances` package in Figure 7. It contains classes that define instances of all elements necessary to define a use case instance (all subclasses of `ObservableInstance`). It also defines a `SystemState` class that is composed of a set of class instances. This means that the state of the system at one moment in the simulation is characterized by the set of values for all objects defined in the requirements. The `clone` operation is used when running one simulation step: to build the new system state, the current state is cloned and modified according to the use case's post-condition. An initial system state has to be provided by the user in order to initiate the simulation.

The second package that is added for simulation is `simulation` (figure 7) that contains the three classes `UCSImulator`, `Scenario` and `ExecutionStep` that implement the simulation mechanism. The `UCSimulator` defines seven operations used to initialize and run the simulation. The simulation can be initialized either with one system state provided by the user (`initializeFromState`) or with a scenario that has been saved from a previous simulation (`initializeFromScenario`). In the latter case, the simulator runs the sequence of use cases specified by the scenario. The current state reached at the end of the sequence is the initial state for the new simulation. The three operations `run`, `runUCInstance`, and `getUCInstances` implement the simulation. The operation `getUCInstances` computes the set of use case instances that can be executed according to the current state of the system (the current state implies the pre-condition for the use case). The operation `runUCInstance` computes the new system state resulting from the execution of a given use case instance. Then it updates the current state with this new state. At last, `run` is the main operation for the simulation. At each step, it calls `getUCInstances`, waits for a user input who chooses the use case to execute among the set of possible use case instances, and calls `runUCInstance` with the chosen instance.

In addition to these two packages, we defined operations in the `Expression` class. The `evaluate` operation checks whether the expression evaluates to true in the context of the system state it receives as a parameter. The `update` operation updates the provided state in order for the expression to evaluate to true. All the packages, classes and operations added into the requirements metamodel have been implemented with Kermeta [9].

5 Related work

Recent tools for requirements analysis tend to define a core model that represents the captured information. Several inputs are used to populate the core model, like constrained natural languages and graphical languages (UML etc.). The core model is then transformed into one or several output models suitable for properties checking tool.(like SPIN [8] used in [10], FMONA [3] used in [2]). The Dwyer patterns [7] are a good example of the need for a unified approach. The intuition behind these patterns is that, there exists a lot of different formalisms (often one formalism for one tool), while the concepts manipulated by these formalisms are restricted. These patterns thus provide a core model for analysis of requirements using temporal logic. In the same way, we have defined our own core metamodel for functional requirements.

Although the implementation of these tools deal with different types of models, they do not use the MDE tech-

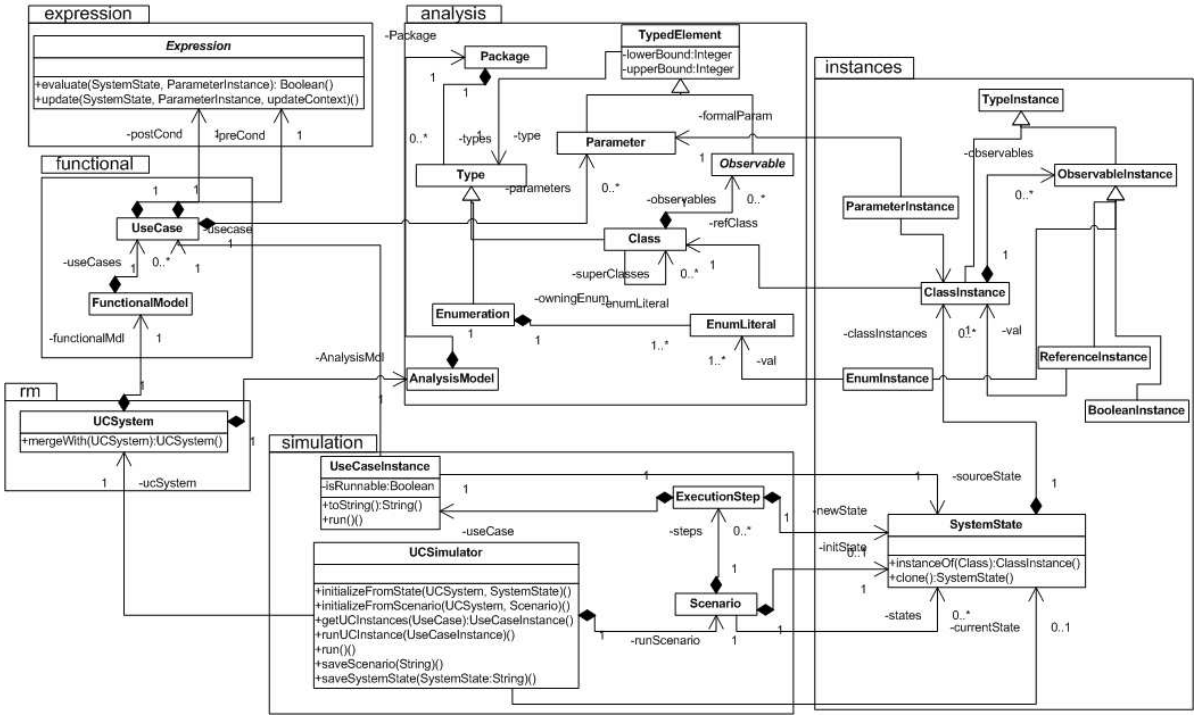

Figure 7. Extended metamodel for requirements simulation

niques. In [6], a metamodel is proposed to capture requirements of real-time system. The models are obtained using syntactical patterns: textual requirements are written, that strictly respect the syntactical patterns. The benefits of such an approach is that the sentence structures that have been chosen for the syntactical patterns impose to disambiguate the requirements (our approach also benefits from this disambiguation). However, the classes of the metamodel are manily composed of attributes of type `String`. In other words, when the requirements are parsed using the syntactical patterns, all the word groups that are not part of a pattern (for example : the subject of an action, or the condition to perform an action) are not interpreted and treated as Strings. That means that the obtained model cannot possibly be programmatically handled for example for a simmulation. Of course the requirements are more easily written than with our approach, but in the other hand they can hardly be validated with automated process, and they cannot either be used for test generation purpose.

Concerning requirements analysis, authors of [10] present an integrated tool suite called SPIDER. It allows users to specify UML Models for analysing temporal behavioural properties of the model. The tool is especially dedicated to analyse systems whose implementation follows the MDA principles. MDA [14] advocates the use of models and model transformations in order to separate business and application logic from the underlying technological platform. The properties are expressed in a constrained natural language (like the DNL of [17]) whose accepted sentences match well the Dwyer temporal logic patterns [7]. In the same way we define a constrained language called RDL and a transformation towards a requirement model, thanks to interpretation patterns. Besides analysis, other works aim at generating test cases from requirements. In particular, several approaches [4, 1, 13] use requirements expressed with extended use cases to generate test cases or at least test objectives. This kind of work shows the benefits that are obtained when the requirements take the form of use cases and emphasizes the need for validated use cases, for example using the simulation mechanism we propose. In the same vein of test generation from requirements, the authors of [2] propose a tool suite called RETNA, which provides analysis and test case generation from requirements expressed in terms of natural language. Similar to our approach, their internal model is based on state machines. The implemented test criterion is robustness (rejection paths), while we implement more possibilities (robustness and nominal behavior). The test criteria we have implemented are explained in [13].

6 Discussion and Conclusion

Model-driven engineering (MDE) offers a new approach for software development, which considers models as first class entities. The work presented here applies it to requirements engineering. We have proposed a metamodel for requirements and extended it to add the semantics with Kermeta. This executable metamodel provides capabilities to model and simulate requirements. We have also developed a series of model transformations that generates a requirement model from a constrained natural language, which is not presented here. The initial intent of this work is to study whether MDE techniques offer good solutions to unify RE activities (agreeing, modelling, simulating...). To evaluate this intuition, the requirements metamodel and the RDL have been experimented to model requirements with THALES Airborne System (TAS) components and France Telecom. The case studies for TAS concerned two systems components of last generation combat aircrafts, of mid-complexity (around 5-15 C++ KLOC). France Telecom studies concerned three services for the LiveBox modem. The goal was to simulate these services and validate functional requirement documents. The following observations were drawn:

- The current metamodel captured most of the concepts needed to express the requirements we had to deal with.
- Simulation is an efficient to reveal underspecifications in requirements. For example, it easily revealed that services that should have been available at one point in the simulation were not available (revealing errors in contracts).

These initial observations are very encouraging to consider our prototype as a good solution to model requirements and simulate the behaviour for validating and agreeing the requirements. These results need to be confirmed with additional case studies. Of course, our metamodel does not allow to capture any requirement : it captures efficiently the dynamic part of the requirements, but not the static part, including the handled data. As future works, we plan to enhance this metamodel (or create another dedicated metamodel) to capture static aspects. We also want to add model transformations that can extract particular views on the model (such as a UML model) and that improve the traceability between the requirements and the implementation.

References

[1] F. Basanieri, A. Bertolino, and E. Marchetti. The cow_suite approach to planning and deriving test suites in UML projects. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, *UML 2002 - The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools. 5th International Conference, Dresden, Germany, September/October 2002, Proceedings*, volume 2460 of *LNCS*, pages 383–397. Springer, 2002.

[2] R. Boddu, L. Guo, S. Mukhopadhyay, and B. Cukic. Retna: From requirements to testing in a natural way. In *RE04 (Requirements Engineering)*, pages 262–271, Kyoto, Japan, 2004.

[3] J.-P. Bodeveix and M. Filali. Fmona: A tool for expressing validation techniques over infinite state systems. In Springer-Verlag, editor, *Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems: Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000*, pages 204–219, 2000.

[4] L. Briand and Y. Labiche. A UML-based approach to system testing. *Journal of Software and Systems Modeling*, pages 10–42, 2002.

[5] F. Budinsky, T. Grose, D. Steinberg, R. Ellersick, E. Merks, and S. Brodsky. *Eclipse Modeling Framework: a developer's guide*. Addison-Wesley Professional, 2003.

[6] C. Denger, D. M. Berry, and E. Kamsties. Higher quality requirements specifications through natural language patterns. In *SWSTE '03: Proceedings of the IEEE International Conference on Software-Science, Technology & Engineering*, page 80, Washington, DC, USA, 2003. IEEE Computer Society.

[7] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In I. C. S. Press, editor, *ICSE*, pages 411–420, 1999.

[8] G. Holzmann. The model checker spin. In *IEEE TSE*, pages 279–294, 1997.

[9] Kermeta. The kermeta project home page, 2005.

[10] S. Konrad and B. H. C. Cheng. Automated analysis of natural language properties for uml models. In *MoDeVa'05 (Model Design and Validation Workshop associated to MoDELS'05)*, Montego Bay, Jamaica, 2005.

[11] D. Lugato, F. Maraux, Y. Le Traon, C. Nebut, V. Normand, H. Dubois, J.-Y. Pierron, and J.-P. Gallois. Automated functionnal test case synthesis from thales industrial requirements. In *RTAS'04*, Toronto, Canada, 2004.

[12] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In *MoDELS'05*, pages 264 – 278, Montego Bay, Jamaica, 2005. LNCS.

[13] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jézéquel. Automatic test generation: A use case driven approach. *IEEE Transactions on Software Engineering*, 2006.

[14] OMG. Mda, 2003.

[15] OMG. Mof 2.0 core final adopted specification., 2004.

[16] OMG. Object constraint language specification, version 2.0, 2006.

[17] R. L. Smith, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil. Propel: An approach supporting property elucidation. In A. Press, editor, *ICSE*, pages 11–21, 2002.

[18] Sommerville. Integrated requirements engineering: A tutorial. *IEEE Software*, 22(1):16 – 23, 2005.